Science of Computer Programming 4 (1984) 211–218
North-Holland

BOOKS

Books should be sent to the editor-in-chief. Selected books, which are within the scope of SCP and are not proceedings, will be reviewed. Others may be mentioned.

Principles of Concurrent Programming. By M. Ben Ari. Prentice-Hall International, London, 1982, 172 pp.

There are two good textbooks on the subject of operating systems which devote a lot of attention to concurrent programming: Brinch Hansen's *Operating Systems Principles* and Holt's et al: *Structuring Concurrent Programming with Operating Systems Applications.* The author of the book under review rightly decided that the time is ripe to offer the readers a textbook entirely devoted to the subject of concurrent programming. The book is appropriate for a semester course on the subject. As the author observes no prerequisites other than computer science maturity are required.

The book consists of seven chapters and an appendix. After two introductory chapters focusing on the nature and origins of concurrent programming the author discusses the problem of mutual exclusion and presents Dekker's solution. The presentation is based on the original Dijkstra's article while more recent papers (like Lamport's bakery algorithm) are relegated to the exercises. This creates an erroneous impression that not much has happened in this area since 1968. Various issues like solutions for n processes, distributed solutions, concurrent reading and writing and solutions for the case when processes can fail are introduced as secondary problems or even not mentioned at all. The only innovation of this chapter is that the correctness proof of Dekker's solution is presented in a rigorous yet informal way using temporal logic.

Next chapter is devoted to the study of semaphores. The presentation is once again based on Dijkstra's original article but this time more references to the recent literature can be found in the text and exercises. In particular simulation of a general semaphore by a split binary semaphore and the cigarette smoker's problem are discussed in the exercises.

Next chapter is about monitors. The presentation is this time based on the original article of Hoare. Unfortunately exercises are more about semaphores than monitors. In particular none of the issues studied after the introduction of monitors—like conditional **wait** or nested monitor calls are even mentioned. Topics discussed so far can be found in one way or the other in the books mentioned before: those of Brinch Hansen and Holt et al. The first novel chapter in this regard is the one about

the ADA rendezvous concept. The presentation is on the other hand very brief and sketchy.

The final chapter is devoted to the presentation of the problem of dining philosophers. Various attempts of solutions and correct solutions are coded using the semaphores, monitors and conditional critical regions. The book concludes with a long appendix in which a listing of a simple system allowing a concurrent execution of simple Pascal programs augmented by semaphores is presented. The system itself is written by N. Wirth.

A novel aspect of the book is the presentation of the proofs of the programs using a semi-formal reasoning based on temporal logic. The proofs are convincing and very well presented. They take in total 9 pages and provide the best informal introduction to the subject of temporal logic one could think of.

Apart from these 9 pages and the short, 16 pages long chapter on the ADA Rendezvous the whole book could have been written some six–seven years earlier. Rare pointers to the literature on the subject written after 1975 do not change this overall impression.

It is a pity that there is no chapter about CSP. This would allow the author to focus more on some of the problems inherent to distributed processing like the problem of distributed termination of Francez only cited in the bibliography. Such a chapter would improve the balance between parts devoted to multiprogramming and distributed processing. Also an introduction of chapter on CSP would allow a critical comparison of two different tools introduced for the same purpose—CSP and the ADA tasking, an approach so successfully exploited in the book in the case of multiprogramming.

On the other hand it should be clearly stated that the book is very well if not exceptionally well written. It makes really a pleasant reading. Regular references to chivalrous esquimos trying to work out appropriate protocols to enter their igloos or to exchange meat for sandwiches help to visualize the addressed problems and to understand the proposed solutions. Writing about concurrent programming is not an easy task and the author succeeded in this domain superbly presenting the material in a perfectly clear and convincing manner. The only objection one could have is that some of the exercises of the form "Study the following program ..." without any additional comments can only frustrate the reader rather than to enlighten him. Studying 'raw' concurrent programs can be really a very difficult task.

Summarizing, this is a very nicely written book which offers a bit too old and uneven view of the subject. Those wishing to get complementary information on the subject are recommended to consult Andrews and Schneider's article "Concepts and notations for concurrent programming", *Comput. Surveys*, **15** (1) (1983).

Krzysztof R. APT
LITP, Université Paris 7
Paris, France

Algorithmic Language and Program Development. By Friedrich L. Bauer and Hans Wössner. Springer, New York, 1982, 497 pp.

or: **Towards a science of programming***

1. Introduction

In a field as young and dynamic as computer science it is surprising that the current main reference and textbook on the subject of computer algorithms were written between 10 and 16 years ago [1, 7]. These sources established algorithms as the basis for programming and the main foundational area of computer science. Based on pragmatic considerations, these sources defined a framework for investigating and presenting algorithms in which each problem is specified informally in English, and each algorithmic solution is specified in a low level ALGOL-like language. Although this framework has proved itself effective, it has four serious shortcomings:

(i) *Problem specification.* Because the problem is not stated in a formal language, it is often lacking in clarity; the only formal statement of the problem is the procedural code that purports to give the solution.

(ii) *Program transformation.* The lack of formal problem specification also precludes the possibility of directly transforming the problem into an efficient procedural solution. Thus, the synthesis of the procedural solution is done in an *ad hoc* way that is largely indifferent to the problem statement.

(iii) *Program correctness.* To ensure the correctness of the solution a formal correctness proof must be stated with respect to the procedural solution, and must necessarily incorporate concrete dictions at the same low semantic level as the solution. This makes the formal proof long, complicated, and unconvincing (see [3]). Thus, the current algorithm framework seems to work best when the correctness of the solution is handled using traditional informal mathematical techniques.

(iv) *Program analysis.* The time and space complexity of the solution is based on a rigorous mathematical analysis of low level structural properties of the solution; e.g., branching and recursion. This analysis is mainly determined completely independently of any design principles used to construct the solution.

No one can doubt that overcoming these difficulties would have a profound effect on algorithms and programming, but the problem is formidable and progress has been slow. Nevertheless, some progress has been made. In the last ten years three major group research efforts have attacked this problem from different points of view. Edinburgh LCF [6] has focussed on logic, SETL [5, 8, 9] on transformation, and project CIP (see the book discussed in the current review) on specification.

* The research on which this review is based was supported by the National Science Foundation under Grant No. MCS-8212936.

The first of these, LCE (Logic for Computable Functions), is an ongoing project at Edinburgh and Cambridge in Great Britain. It has produced a general purpose functional programming language based on LISP called ML, and a proof language based on a logic due to Dana Scott [10]. This project places special importance on proofs of program properties, including the correctness of both programs and transformations.

The SETL project was active for over 10 years under the direction of J.T. Schwartz at the Courant Institute of New York University. It produced the high level set theoretic programming language, SETL, and a working optimizer. Like the original FORTRAN project of John Backus and the optimization work of John Cocke, the SETL project was an engineering effort that stressed automatic program optimization. The current SETL optimizer implements transformations for the selection of conventional storage structures and aggregations for set and map variables.

The third research project is CIP (Computer-aided, Intuition-guided Programming), which has been directed by F.L. Bauer at the Technical University of Munich since 1976. They have produced a powerful ALGOL-like programming language for specifying abstract problems as well as their associated low level implementations. These implementations are derived directly from their problem statements using manually specified correctness preserving transformations.

2. Algorithmic language

Many of the essential ideas developed in project CIP are reported in the new book, *Algorithmic Language and Program Development*, by Friedrich L. Bauer and Hans Wössner in collaboration with Helmut Partsch and Peter Pepper. This research monograph is based on the authors' lectures in graduate courses as well as on their research from the CIP project. It is relatively self contained, and is written in an informal style: hard proofs are omitted, and explanations that are outside the scope of the book are suggested by well selected examples, incorporated into exercises, or found in the numerous references that are provided. The authors reflect a mature scholarship with a unique historical perspective, a comprehensive discussion of background sources, an extensive bibliography, and a thorough discussion of the subject matter.

The main focus of the book is on a new programming language, called Algorithmic Language (abbreviated AL in this review). AL is exemplified by a rich semantics, notational clarity, and extensibility. It is a strongly typed language that can specify all the semantic levels of description from the abstract level of problem specification down to the concrete level of machine implementation. The notations are based on ALGOL 68 supplemented with guarded commands, logical quantifiers ($\forall$, $\exists$), a primitive data type for commutative semigroups, an elaborate facility for defining and making use of polymorphic data types, and much more. Procedures can be both

parameters and results, providing the full power of Lambda Calculus. AL has an arbitrary selection operator, *some* x: $p(x)$, which has the value of some arbitrary element satisfying the predicate p. AL also has a unique selection operator, *the* x: $p(x)$, which yields the value satisfying p when there is only one such value, and the 'undefined' value otherwise.

These selection operators, which Bauer and Wössner attribute to classical logic, provide the language with an enormous abstract expressiveness well suited to problem specification. Numerous illustrations of these and other language features, especially the abstract data type facility, demonstrate convincingly that AL can easily specify all kinds of data and computational structure at all semantic levels from the mathematical level down to the machine level.

These features of AL also seem well suited to a formal program construction methodology by transformation. The mathematical semantics and strong typing support the verification of abstract problem specifications. The efficient implementation can then be proved correct, because it is derived directly from the initial specification by correctness preserving transformations. However, the discussion of program derivation by transformation is less developed than the discussion of program specification.

At the beginning of the book the authors claim that AL is "a programming language designed according to the principles of program transformation". Although it is a 'remarkable' intuitive insight that a programming language such as AL should be designed to facilitate transformations, the claim that AL was actually designed like that is not supported in the book. Nor is it likely, given the current state of research, that this insight could be developed in a substantial way. This is because a truly viable theory of program transformation would provide the basis for a theory of algorithm design. But as I said earlier, the current field of algorithms is based on a framework 10 years old for which little is know about the aspect of 'design'. Certainly, Dijkstra [4] has some very compelling informal ideas about algorithm design, but he is pessimistic about capturing these ideas within a formal transformational system, and he even seems reluctant to commit himself to anything but the simplest language for illustrating these ideas.

Bauer and Wössner also understand how difficult algorithm design is, and they express pessimism about the possibilities of finding a small, relatively complete basic collection of powerful algorithmic transformations. "The amount of intuition required in (algorithm development) cannot be over-estimated." Instead they provide a basic repertoire of low level transformations that include unfolding, folding, abstraction, embedding, and recursion removal. And they place responsibility on the user to freely combine these primitive transformations in order to implement more meaningful computation and data type mappings.

Because these basic transformations are so low level, it is not likely that they could have been a real influence on the design of AL. It is, of course, possible that the more meaningful higher level transformations suggested in numerous examples of program derivation could have had an influence on the design of AL. But since

no general taxonomy of high level transformations is provided, the link between transformation and language is left only implicit.

The beginnings of such a taxonomy are suggested by the transitions within a scheme for deriving programs through four successive development stages:

(i) *Pre-algorithmic.* This is a problem statement formulation expressed using either an arbitrary or unique selection operator.

(ii) *Functional.* The algorithm in this stage is expressed as an applicative expression with explicit recursion.

(iii) *Procedural.* At this point intermediate variables are introduced, and recursions have been removed in favor of iterations, mainly while loops.

(iv) *System.* Efficient use of storage structures with pointer access mechanisms prevails.

An informative example of GCD on pp. 452–454, illustrates the preceding program development scheme.

The 4 stages of program development proposed here are new and interesting. At present they seem to be based on programming language semantics rather than on any of the few known algorithm design principles; e.g., the use of dynamic data structures, balancing, path compression, use of depth first search orderings, and so forth. The requirement to use explicit recursion in the functional stage could probably be relaxed to allow functionally equivalent iterative expressions based on closure operations. As the authors point out, the system stage needs further development.

This 4 stage scheme and the primitive transformations discussed in this book are still too rudimentary to easily capture the more informal reasoning used by Dijkstra [4] to derive algorithms. But the authors provide enough evidence that some future version of AL would come much closer to achieving this ambitious goal. Their current collection of primitive transformations forms a powerful general purpose transformational assembly language. Such a language could evolve into a more significant language in the same way that the main language component of AL evolved from ALGOL. As primitive transformations are used to implement higher level constructs, those constructs that are seen to be most highly applicable can be captured as primitives in a higher level language. The development of such a language could eventually lead to a theory of algorithm design.

3. Presentation

The book is an English translation by the authors of an original text that they wrote in German. Although the authors are native Germans, the writing style of this English translation is surprisingly good. The interjection of side remarks (set off in a very small font) in which the authors comment on their own material is novel, and also enables the authors to tell a story in parallel with the main subject matter. This story is in large part the personal experience of Professor Bauer (see also [2]), who participated in the historical development of Europen computer science from ALGOL 58 to ALGOL 60 to ALGOL 68 to AL.

The book is rich in historical references, especially to the early contributions of European computer scientists. It is worth mentioning a few of these contributions, which are not widely known among Americans. In 1952 the Swiss mathematician, Heinz Rutishauser, who worked with the German computer pioneer, Konrad Zuse, used unfolding transformations to unroll recursive definitions. The reader might be surprised to learn that Bauer and Klaus Samelson invented the stack mechanism for implementing recursive ALGOL in 1958, and they actually hold a patent on it.

In such a young field as computer science in which seminal contributions can be easily lost to folklore, Bauer and Wössner have provided an invaluable service by preserving the origins. The book would have benefitted from a similar concern for important contemporary research that is alternative to their own approach. There should be a more serious characterization and comparison with SETL and some discussion of LCF. Unfortunately also, articles coming out of the SETL and LCF projects fail to make reference to the contributions of project CIP.

The authors are well aware that there is too much to say in too brief a space. The book is packed solid with ideas and terminology that are usually well presented, but are sometimes presented too informally and without proper motivation. For example, on p. 128 it says "looked at mathematically the entirety of selections forms a category." There is no definition of 'category' and the significance of this observation is not explained. Ershov's transformational technique called 'mixed computation', which arouses the reader's curiosity, is not clearly explained; fortunately, a good reference is provided.

The format of the book is exemplary. The publisher has accommodated all the material by using a fairly small font for the main text, and small margins. Very attractive art figures introduce each chapter. The Table of Contents and Index are complete and useful. As an additional convenience the book contains a glossary of terms, separate indexes for AL language features, and separate bibliographies for the main scholarly references and references to programming languages mentioned in the text.

4. Conclusion

This book has broad relevance to the theory of algorithms, and makes foundational contributions to the science of programming, particularly in the area of specification. Although further development of the logic and transformational components of AL would be necessary to create a strong impact on the areas of software engineering and algorithms, this work by itself will undoubtedly have an impact on the future of programming languages.

Robert PAIGE
Rutgers University
New Brunswick, U.S.A.

References

[1] Aho, Hopcroft, Ullman. *Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).
[2] F.L. Bauer, Between Zuse and Rutishauser—The early development of digital computing in Central Europe, Technical Report 7629, TUM, Institut für Informatik (1976).
[3] R.A. DeMillo, R.J. Lipton and A.J. Perlis, Social processes and proofs of theorems and programs, *Proc. 4th ACM Symposium on Principles of Programming Languages* (1977) 206–214.
[4] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).
[5] S. Freudenberger, J. Schwartz and M. Sharir, Experience with the SETL optimizer, *ACM TOPLAS* **5** (1) (1983) 26–45.
[6] M. Gordon, A. Milner and C. Wadsworth, *Edinburgh LCF* (Springer, Berlin, 1979).
[7] D.E. Knuth, *The Art of Computer Programming*, 3 Volumes (Addison-Wesley, Reading, MA, 1968–1972).
[8] E. Schonberg, J.T. Schwartz and M. Sharir, An automatic technique for selection of data representations in SETL Programs, *ACM TOPLAS* **3** (2) (1981) 125–143.
[9] J.T. Schwartz, Automatic data structure choice in a langauge of very high level, *Comm. ACM* **18** (12) (1975) 722–728.
[10] D.S. Scott, Lattice theoretic models for various type-free calculi, *Proc. 4th International Congress in Logic, Methodology, and the Philosophy of Science* (1972).